Handling Exceptions in Programs with Hidden Concurrency: New Challenges for Old Solutions

Alcides Fonseca, Bruno Cabral
Universidade de Coimbra
Coimbra, Portugal
{*amaf,bcabral*}*@dei.uc.pt*

***Abstract*—Multi-core processors are present in everyone's daily life. Consequently, concurrent programming has re-emerged as a pressing concern for everyone interested in exploring all the potential computational power in these machines. But, the emergence of new concurrency models and programming languages also brings new challenges in terms of how one can deal with abnormal occurrences, much due to the heterogenous parallel control flow. Unexpectedly, sequential Exception Handling models remain as the most used tool for robustness, even in the most recent concurrent programming languages. Though, the appearance of more complex models, such as programming languages with implicit concurrency, might pose a challenge too big for these sequential mechanisms. In this article we will provide evidences why such models are not generally suited to deal with faults in programs with implicit concurrency and, in the light of more recent advances in concurrent Exception Handling, we will discuss the attributes of a model for addressing this problem.**

I. Introduction

The development of new concurrency models and concurrent programming languages has become one of the most active research fields in Computer Science of the last decade. At the root of this growing interest is the need for programs that are able to efficiently explore the computational power available on modern multi-core processors. But, concurrency introduces new challenges for systems development, among which Exception Handling (EH) is a main concern. More than confining developers to the usage of sequential *try-catch* type EH techniques, language designers face the challenge of integrating EH into a new and complex environment in a way that respects the structure of programs and its goals. And, despite the existence of sound concurrent and distributed EH models(e.g., [1], [2]), the traditional and sequential exception handling model remains as the most used tool for robustness in concurrent programming. This is still true in more concurrency-oriented languages such as Scala, Fortress, X10, JCilk, and Erlang, among others. But, such hegemony may end soon. The reason for this downfall is the new interest in programming languages and models capable of generating concurrency without an explicit structure or control by the programmer [3], [4] (section II). As we will show in this article, such lack of control makes the usage of sequential EH impossible and the integration of existent concurrent solutions too costly. Furthermore, since these languages are very recent and still an open research topic, all aspects related with EH remain unexplored.

The main contributions of this work are:

- Identification of the problems which make the use of sequential *try-catch* type EH mechanisms undesirable for programming languages with implicit concurrency (section III);
- Discussion of the pitfalls of using concurrent EH techniques for programs written in such programming languages (section IV.)

II. Programming Languages with Implicit Concurrency

Writing concurrent programs is a complex and error-prone task. The main difficulty arises from the fact that we are used to think about programs sequentially. Moreover, traditionally developers are not taught to reason about programs concurrently. Consequently, parallel programming is almost always an afterthought. Programmers have to specify, control and tune parallel execution explicitly and effectively. For instance, programmers must identify and control all possible memory-access interferences (e.g., races). Such an approach to parallelism does not scale well with the increasing complexity in programs.

During the last decade, many programming languages designers have been working towards decreasing the complexity of concurrent programming. Many new models and languages have emerged from this effort. In this work, we will focus in languages with implicit concurrency, i.e., languages where the programmer is allowed to write code very similar to what he or she would write for sequential programs, but which will execute concurrently "under-the-hood". We have already mentioned a couple of these language [3], [4], but we will be focusing in the approach which, in our opinion, seems to illustrate better the similarity with sequential code and the code that the programmer actually produces.

In Æminium[4], programs are parallel from inception and developers do not control or define parallelism explicitly. Even the execution order of the instructions in the code is automatically decided by the compiler and runtime. Developers focus on the definition of Access Permissions for the objects in the code. These permissions allow the compiler to understand the Data Dependencies between all instructions in the code, which it uses to derive a data-flow graph of the program and allow the parallel execution of all independent tasks in the graph.

Access permissions are special annotations introduced into method signatures which describe the restrictions for accessing the objects being referenced. We will consider four kinds of permissions: *unique*, *shared*, *immutable* and *none*. *Unique* means that there can only be one reference to the ”tagged” object in every moment of the program execution. *Shared* means that there may be two or more references to an object, but accesses to it must be performed in consistent way (e.g., mutual exclusion). *Immutable* expresses the fact that the object is not modified, thus several tasks can access it in parallel. *None* means that there is no reference to the object. Access permissions are written by the programmer in the source code of the program, they can be optionally used in local variables, but are mandatory in method signatures.

```
state UpperCaseFileConverter {
    method void convert(immutable String inputName, immutable String
          outputName) [none] {
        val inputFile = java.io.File.new(inputName);
        val outputFile = java.io.File.new(outputName);
        val reader = java.io.BufferedReader.new(java.io.FileReader.new(inputFile
              ));
        val writer = java.io.BufferedWriter.new(java.io.FileWriter.new(outputFile)
              );

        translate(reader, writer);

        inputFile.close();
        outputFile.close();
    }
    method void translate(unique BufferedReader reader, unique BufferedWriter
          writer) [none] {
        while (reader.hasAvailable()) {
            var newLine = translateLine(
                    readLine(reader)
                );
            writeLine(writer, newLine);
            Logger.log(”Converted ” + newLine);
        }
    }
    method immutable String readLine(shared BufferedReader reader) [none] {
        reader.readLine();
    }
    method void writeLine(shared BufferedWriter writer, immutable String
          newLine) [none] {
        writer.write(newLine);
        writer.flush();
    }
    method immutable String translateLine(immutable String oldLine) [none] {
        oldLine.toUpper();
    }
}
```

Listing 1. Example of a FileConverter in Æminium

The program in Listing 1 performs the conversion of the text in a file to upper case and then saves the result in a new file. This example will help us understand how Æminium works. It is important to notice that several parts of this program will execute in parallel. But, without the permissions in the code it would actually be very difficult to distinguish this code from regular sequential code. In particular, the method `translateLine` can be invoked in parallel with other instances of itself (even when applied to the same string) because there is no reference to `this` and the only parameter present is *immutable*. On the other hand, the method `readLine` cannot be invoked in parallel with itself using the same `reader`, because access is *shared* and therefore needs to be synchronized. `readLine` and `writeLine` can be executed in parallel because there is no access to `this` and they do not share any parameter. Besides these restrictions defined by the access permissions, the compiler also guarantees the order between parallel executions, so that the output is the same as if the program was sequential. This is done using a data-flow approach.

In this example, the mentioned restrictions simply mean that reads and writes of files are sequential and ordered due to the *Unique* permission, but the actual execution of the `translateLine` can be parallelized. For instance, at a given time the program may have read 10 lines and it is processing those 10 lines without having written anything to disk yet.

III. Sequential EH and Implicit Concurrency

In the example in Listing 1, IO operations are not guaranteed to always execute correctly. For instance, open, read and write operations may raise exceptions if some problem occurs. In this section we will consider several abnormal scenarios using Æminium code examples.

```
method void translate(unique BufferedReader reader, unique BufferedWriter
      writer) [none] {
    try {
        while (reader.hasAvailable()) {
            try {
                newLine = translateLine(readLine(reader));
            } catch (CouldNotTranslateEnconding e) {
                newLine = translateLine(readLastLine(reader, ’utf8’));
            }
            try {
                writeLine(writer, newLine);
                Logger.log(”Converted ” + newLine);
            } catch (DiskIsFullException e) {
                clearTemporaryFiles();
                retry;
            } catch (IOException e) {
                Logger.log(”Could not write on the output file.”);
                break;
            }
        }
    } catch (IOException e) {
        Logger.log(”Line ’” + newLine + ”’ was not converted because of ” + e.
              toString() );
        outputFile.delete();
    }
}
```

Listing 2. Handling of Exceptions in File Converter if it was Sequential

Starting with the previous example, if we consider it to execute in a pure sequential way, the code in Listing 2 would suffice for recovery under several abnormal conditions when replacing the original `translate` method. Please note that for the programmer (and for anyone reading the code) this recovery code would look suitable if the code executed the way it is actually written, which is not the case here. In Æminium, we do not know exactly how this method will be executed.

In Æminium, considering that the `log` operation can be called multiple times inside the loop and that those calls are executed in parallel, several things can go wrong. These are situations where this code could fail to recover from abnormal situations in Æminium due to its concurrent nature:

- *Occurrence of an error reading from the file* - In this case the output file should be deleted and the operation aborted. However in a implicit concurrent language, one thread might catch an error when reading a line and handles it by deleting the output file while another thread is finishing the translation of the previous line and writes to the same file shortly after. This results in an inconsistent program state.
- *Occurrence of an error translating a line due to string encoding problems* - It is possible to read the line again (using `seek` for instance) with another decoder. In a implicit concurrent language this can not be done because the last line read might not be the one causing problems. The translation of each line happens in parallel with a lock protecting the file reads. It is possible that the next iterations of the cycle have already read their lines before the faulty translation happens.
- *Occurrence of an error writing to the file* - The success message should never be written to the logger, and the operation would be aborted. In a concurrent setting, the log instruction might still execute in parallel with `writeLine` and incorrect information would be logged. This means that even though the `writeLine` fails, the log might have already completed or still be running in parallel. Even if both lines were inside a protected block, it would not be possible to predict which one would finish first.
- *Occurrence of an error writing to the log* - It is not clear how exceptions should be thrown and handled. On one hand, since the calls are parallel, each one should raise one exception. On the other hand, from the perspective of the programmer, there is only one single error that affects the parallel execution several times.

In any of the previous cases, to correct the issue the programmer has to prevent future loop iterations from occurring, pause the program, correct the problem and retry. Alternatively, he or she could enable each parallel execution to try to restore the logger itself. It is very hard to know exactly what has already been written to the output file, what has been translated or is being translated. Programmers do not know what other operations can be in execution simultaneously.

It is also feasible that the program will continue to execute, consuming resources, just to fail all the operations that are already in execution. Since this is a very simple example, we can guess almost all the possibilities, but in more complex programs it would be impossible to know exactly what is executing and how the isolation between concurrent executions is broken by one exceptional occurrence.

To summarize, an implicit concurrent language poses the following challenges when we attempt to apply sequential exception handling techniques to it in the event of an abnormal occurrence:

- It is not possible to assert the precise state of the program when writing the code. Even if no recovery is attempted and the program is aborted, the system would have an hard time logging the precise conditions of when the problem occurred.
- It is not possible to know what is executing in that moment when writing the code.
- It is not possible to know what has executed until that moment when writing the code.
- It is not possible to know the impact in concurrent executions or notify them of the problem.
- It is not simple to revert the program to a ”known” state.

One can use a conservative approach, assigning *Unique* permissions to as many objects as possible in order to ”sequentialize” execution and make sequential EH work. But, this would break the fundamental objective of this paradigm, which is extracting as much concurrency as possible from the code.

IV. Can Concurrent Exception Handling Approaches Help?

Concurrent EH has been a very active research topic for several decades now. And, although one can argue that there is no *definitive* solution, several techniques have been very successful in particular domains (e.g., Atomic Actions, Conversations, Coordinated Atomic Actions, the Guardian model, among others). Unfortunately, a common requirement to all these approaches is that they need concurrency to be explicit in the source code, for instance, with threads and locks. This is something which implicit concurrent languages do not have since the scheduling of code among the available threads only happens at runtime. Therefore such mechanisms, without any substantial modification, cannot be used in these languages.

The solution for the problem can be the re-engineering of the runtime of these languages. An approach is to reduce the number of possible states by introducing barriers or checkpoints in some sections of the code[5]. This would allow programmers to handle exceptions knowing that all the code before the last checkpoint executed correctly and will only need to consider the possible states after the last checkpoint. The frequency of checkpoints in a program would lead to a difficult trade-off. On one hand, less frequent checkpoints would lead to more possible alternatives being considered, and more possible states for the programmer to consider. On the other hand, more frequent checkpoints would lead to a more sequential code with less parallelism.

Another approach to consider is to revert the program to a previous state, in which it is possible to recover. When a program throws an exception, the program can stop the current parallel execution and rollback to the most recent common state. This stability is defined by the non-existence of running code that can have potential side-effects on other running code. On a stable state, it would be possible to execute the error-recovery code defined by the programmer. Inspiration can come from the work of Lanvin et all[6], who proposed the concept of *reconstructor*: a counter-action for each operation that can take the program from the state left by that operation to the state it was in before. Having this undo semantics for all methods in a program it is possible to revert to a previous state. The drawback of this technique is that the *reconstructor* must be programmed manually. This means that for every method, programmers must write another one for reverting the state.

Software Transactional Memory (STM) is an alternative for achieving the same goal these days. A way of implementing the system is making protected blocks atomic[7], using STM. Operations done inside a `try-catch` are either all completed or none is. And if there is an exceptional occurrence, there is a `catch` clause that able to restore the system. Although the behavior is what we expect, STM solutions today are still very expensive. Furthermore, if the exception is caused by external factors (writing to a file and other IO operations), the system cannot automatically recover from those, which also poses a problem. For instance, the authors of the Atomic Boxes mechanism [8] report slowdowns between 10 and 1000 times for simple programs featuring sorting algorithms such as *quicksort* and *bubblesort*. Atomic Boxes expand the previous work by isolating atomic blocks. These blocks can have dependencies, something in common with languages such as Æminium, and if there is an exception in one block, all the dependent blocks stop executing and all threads execute the recovery code.

Another alternative would be for the system to automatically handle the most common exceptions[9]. This way the programmer would not have to write EH code itself. On the short-side, this approach suffers from the same problems of the previous atomic models and it is not be suitable for exceptions resulting from bad program logic.

V. Conclusions

In this article we have identified the problems that arise when attempting to use sequential EH techniques in programs with implicit concurrency. We also concluded that such programs would be unable to use concurrent EH techniques because they lack any kind of explicit identification of concurrent structures (e.g., threads) in the code. Furthermore, we also described how some successful approaches for dealing with problems in concurrent settings could be used to improve the system and allow the inclusion of recovery strategies. On this last topic we concluded that none of the described approaches would be totally successful or desirable. In conclusion, EH in languages with implicit concurrency is a challenging new problem.

Acknowledgment

This work was partially supported by the Portuguese Research Agency FCT, through CISUC (R&D Unit 326/97) and the CMU—Portugal program (R&D Project Æminium CMU-PT/SE/0038/2008).

References

[1] V. Issarny, "An exception handling model for parallel programming and its verification," *SIGSOFT Softw. Eng. Notes*, vol. 16, pp. 92–100, September 1991.

[2] R. Campbell and B. Randell, "Error recovery in asynchronous systems," *IEEE Transactions on Software Engineering*, vol. 12, pp. 811–826, 1986.

[3] S. Marlow, R. Newton, and S. Peyton Jones, "A monad for deterministic parallelism," in *Proceedings of the 4th ACM symposium on Haskell*. ACM, 2011, pp. 71–82.

[4] S. Stork, P. Marques, and J. Aldrich, "Concurrency by default: using permissions to express dataflow in stateful programs," in *Proceeding of the 24th ACM SIGPLAN conference companion on Object oriented programming systems languages and applications*. ACM, 2009, pp. 933–940.

[5] L. Gesbert, F. Gava, F. Loulergue, and F. Dabrowski, "Bulk synchronous parallel ml with exceptions," *Future Generation Computer Systems*, vol. 26, no. 3, pp. 486 – 490, 2010.

[6] D. Fernández Lanvin, R. Izquierdo Castanedo, A. Juan Fuente, and A. Fernández Álvarez, "Extending object-oriented languages with backward error recovery integrated support," *Computer Languages, Systems & Structures*, vol. 36, no. 2, pp. 123–141, 2010.

[7] C. Fetzer and P. Felber, "Improving program correctness with atomic exception handling," *Journal of Universal Computer Science*, vol. 13, no. 8, pp. 1047–1072, 2007.

[8] D. Harmanci, V. Gramoli, and P. Felber, "Atomic boxes: Coordinated exception handling with transactional memory," *ECOOP 2011–Object-Oriented Programming*, pp. 634–657, 2011.

[9] B. Cabral and P. Marques, "A transactional model for automatic exception handling," *Computer Languages, Systems & Structures*, vol. 37, no. 1, pp. 43–61, 2011.